VHDL Coding Style Guidelines

From Effective Coding with VHDL - Principles and Best Practice
Ricardo Jasinski, MIT Press, 2016

Version 1.1, August 2017

This document contains selected guidelines and recommendations about coding style from the book *Effective Coding with VHDL – Principles and Best Practice*. It can be used as a coding standard for new projects or by design teams that do not have yet such conventions.

Each recommendation is accompanied by a rationale and references to sections in the book where the topic is explained in more detail. The book also presents a detailed discussion of the fundamental design principles related to writing source code, such as modularity, abstraction, and hierarchy, and explains how they can be used to improve desirable qualities of the code such as readability, maintainability, testability, and reuse.

This work is constantly evolving, and feedback about it is always welcome. To reach the author, please access the Effective VHDL Coding blog at effectivecodingwithvhdl.wordpress.com.

Contents

1. Code Formatting and Layout

The way we lay out statements and other constructs on the screen can make our code much easier to work with. A good layout helps us see scopes, emphasizes the relatioship between statements, and highlights the differences and similarities between lines of code. The guidelines presented here help write code that is easier to read, understand, and maintain.

Indentation

18.3 p 471

Indentation is the addition of blank space before each line of code to convey information about scopes, nesting, and subordination between statements. For a detailed discussion of indentation and an example of the basic indentation algorithm, see section 18.3 on page 471.

R1 Always indent by multiples of a standard amount. 18.3 p 473

Rationale To ensure that the code layout accurately matches its structure, the amount of leading whitespace should always be a multiple of the standard indentation mount. If two regions in the code are indented with the same distance from the left margin, we should be able to trust that they are at the same depth in the logical structure of the code.

The recommended indentation amount in this coding standard is four spaces. This means that each line of code should be preceded by an integer multiple of four whitespaces – 0, 4, 8, 12, and so on. An indentation amount of four spaces is enough to clearly show the code structure, and it still leaves enough room in the line for statements and declarations.

R2 Always indent the code using space characters, not tabs. 18.3 p 475

Rationale This guarantees that the code will look the same in every editor or tool. The width of a whitespace character is always the same, whereas the width of a horizontal tab is configuration-dependent. To save some typing, configure your editor to output the right number of white spaces when you press the tab key. This configuration is usually named “indent using spaces” or “use soft tabs.”

R3 Do not use any indentation scheme that depends on the length of identifiers (names). 18.5.1 p 482

Rationale A somewhat popular style is to indent the continuation line at the same level of the first argument in the previous line:

```
paddle <= update_sprite(paddle, paddle_position.x,
                        paddle_position.y, vga_raster_x,
                        vga_raster_y, true);
```

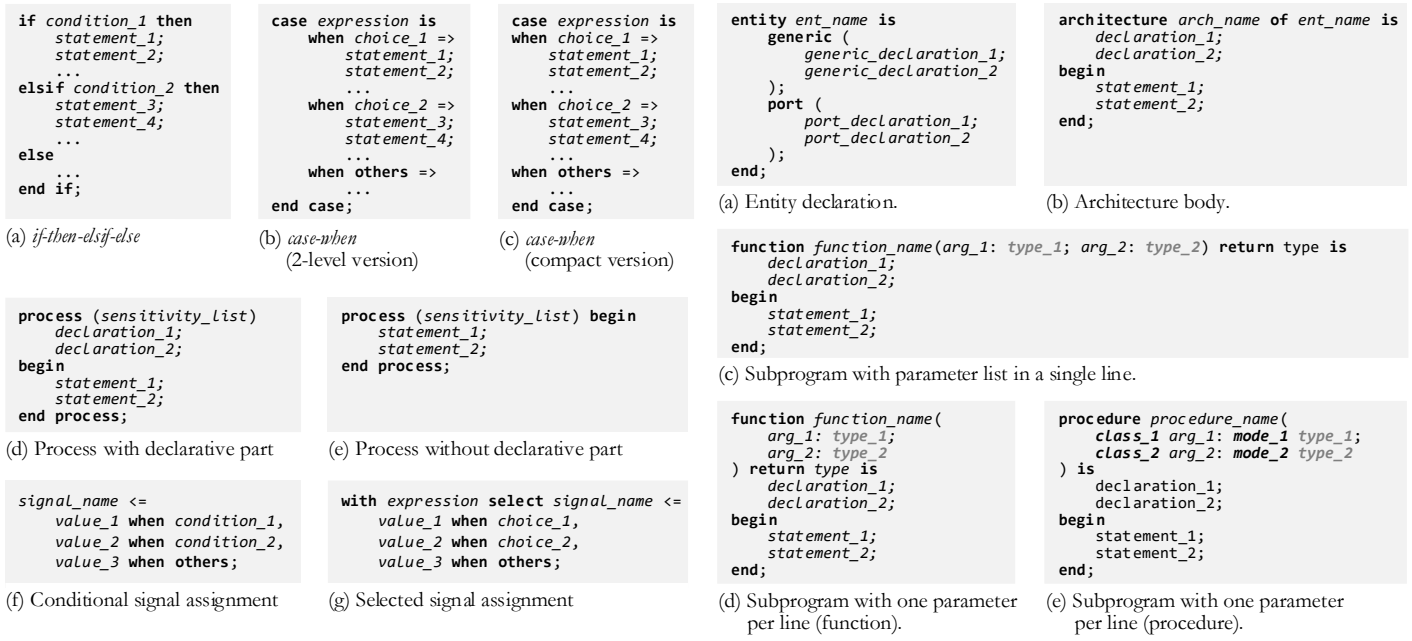
This kind of hanging indent is an example of a style in which aesthetics conflicts with more objective layout goals. It may be aesthetically pleasing, but it is a maintenance headache. It has the following problems:

- It is hard to maintain. If we rename `paddle` or `update_sprite`, we will break the indentation of all subsequent lines, which will have to be manually realigned. These names might be used in several files, so we should hunt for all the occurrences and fix the indentation around them. Because renaming objects and routines is highly recommended to improve the code readability, any layout technique that hinders this practice should be avoided.
- It undermines the strength and consistency of the indentation scheme and its appearance. Instead of always being indented by the standard amount, the names of functions and objects also determine the distance from the left margin, reducing the regularity of the indentation outline.
- It is antieconomic. This kind of layout wastes a lot of space (all the area underneath the signal name and the function name, in the example) and leaves less room for actual code in each line.

In practice, this kind of layout is less helpful than it seems. If you feel tempted to use it (or any form of layout that is offset from the middle of a line), then try instead the technique of emulating pure blocks described in section 18.2.2.

R4 When in doubt, follow the indentation templates provided here as example. 18.3 p 475

Rationale The indentation templates from Figures 18.7 and 18.8 (reproduced below) follow the recommendations from this style guide and comply with the basic indentation principles and algorithm.



```
if condition_1 then
    statement_1;
    statement_2;
    ...
elsif condition_2 then
    statement_3;
    statement_4;
    ...
else
    ...
end if;
```

(a) *if-then-elsif-else*

```
case expression is
    when choice_1 =>
        statement_1;
        statement_2;
        ...
    when choice_2 =>
        statement_3;
        statement_4;
        ...
    when others =>
        ...
end case;
```

(b) *case-when* (2-level version)

```
case expression is
when choice_1 =>
    statement_1;
    statement_2;
    ...
when choice_2 =>
    statement_3;
    statement_4;
    ...
when others =>
    ...
end case;
```

(c) *case-when* (compact version)

```
process (sensitivity_list)
    declaration_1;
    declaration_2;
begin
    statement_1;
    statement_2;
end process;
```

(d) Process with declarative part

```
process (sensitivity_list) begin
    statement_1;
    statement_2;
end process;
```

(e) Process without declarative part

```
signal_name <=
    value_1 when condition_1,
    value_2 when condition_2,
    value_3 when others;
```

(f) Conditional signal assignment

```
with expression select signal_name <=
    value_1 when choice_1,
    value_2 when choice_2,
    value_3 when others;
```

(g) Selected signal assignment

Figure 18.7
Indentation templates for VHDL statements.

```
entity ent_name is
    generic (
        generic_declaration_1;
        generic_declaration_2
    );
    port (
        port_declaration_1;
        port_declaration_2
    );
end;
```

(a) Entity declaration.

```
architecture arch_name of ent_name is
    declaration_1;
    declaration_2;
begin
    statement_1;
    statement_2;
end;
```

(b) Architecture body.

```
function function_name(arg_1: type_1; arg_2: type_2) return type is
    declaration_1;
    declaration_2;
begin
    statement_1;
    statement_2;
end;
```

(c) Subprogram with parameter list in a single line.

```
function function_name(
    arg_1: type_1;
    arg_2: type_2
) return type is
    declaration_1;
    declaration_2;
begin
    statement_1;
    statement_2;
end;
```

(d) Subprogram with one parameter per line (function).

```
procedure procedure_name(
    class_1 arg_1: mode_1 type_1;
    class_2 arg_2: mode_2 type_2
) is
    declaration_1;
    declaration_2;
begin
    statement_1;
    statement_2;
end;
```

(e) Subprogram with one parameter per line (procedure).

Figure 18.8
Indentation templates for VHDL entities, architectures, and subprograms.

Whitespace

R5 Organize your code into paragraphs containing a set of logically related statements or declarations. Always leave a blank line between paragraphs. 18.4 p 478

Rationale This gives the reader a cue when a new step is reached, a subtask is completed, or the main subject changes. It is usually a good idea to precede each paragraph with a summary comment:

```
-- Get character bitmap from font ROM
ascii_code := character'pos(display_char);
char_bitmap := FONT_ROM(ascii_code);

-- Get pixel value from character bitmap
x_offset := pixel_x mod FONT_WIDTH;
y_offset := pixel_y mod FONT_HEIGHT;
pixel := char_bitmap(x_offset)(y_offset);
```

R6 For punctuation symbols that are also used in English, use the same spacing as you would in regular prose. Specifically, never add a space before a comma, colon, or semicolon, and always add a space after these symbols. 18.6 p 483

Rationale Many of the delimiter characters (commas, semicolons, parentheses, etc.) are familiar as normal punctuation marks, so it is distracting to see them in code used differently from normal text. This rule is simple to follow, easy to remember, economic, and consistent. Moreover, leaving unnecessary spaces around these delimiters calls too much attention to symbols that do not have any relevant meaning and exist only to satisfy the language syntax:

```
function add ( addend : signed ; augend : signed ) return signed ;  -- Too much extraneous whitespace; not recommended
```

The following example uses only the necessary amount of whitespace to provide visual separation between the important elements. The punctuation looks more familiar because it is used as in regular prose:

```
function add(addend: signed; augend: signed) return signed;  -- Spacing as in regular prose; recommended
```

R7 For parentheses, use the same spacing as you would in regular prose, with two exceptions:

- Never add a space between a routine name and the parentheses around its parameter list;
- Never add a space between the name of an array and the parentheses used for accessing its elements.

18.6 p 483

```
rising_edge(clk)       to_slv(bv)    -- OK, no space after function name
y(i) <= a(i) + b(i);                 -- OK, no space after array name
```

These are established conventions in virtually any programming language. However, keep in mind that language keywords are not function names, so there should always be a space between a keyword and an opening parenthesis:

```
if (cond)    process (cond)    port map (...)   -- OK, space used after keywords
if(cond)     process(cond)     port map(...)    -- NOT OK, missing space after keywords
```

R8 For the remaining delimiters, use the following rules:

Leave no whitespace around these delimiters	`.` (dot) `'` (apostrophe)
Leave one whitespace on either side of these delimiters	`+` `-` `*` `/` `**` `&` `|` `:=` `=>` `=` `/=` `>` `>=` `<` `<=` `??` `?=` `?/=` `?<` `?<=` `?>` `?>=` `<<` `>>` `<>`

Rationale The dot and apostrophe are not used in VHDL as punctuation symbols. The dot is used in selected names to join the parts of a hierarchical name, and the apostrophe is used in attribute names to join a prefix and an attribute designator. In both cases, it makes more sense to keep the two sides close together than to set them apart.

The remaining delimiters are mostly operators used in expressions, where leaving whitespace on either side generally improves the readability.

Exception For mathematical operators, it may be OK to remove the whitespace around a delimiter as a way of indicating that it has a stronger precedence than the others used in an expression. However, parentheses are still the recommended choice to make the precedence of operators more evident.

R9 Use vertical whitespace to separate sections of the code.

Rationale Just like we can use blank lines to group statements into paragraphs of code, we can use them to separate larger sections of code in a source file. As a rule, leave a blank line before and after every process, function, and design unit in a file.

Vertical whitespace is a cleaner and preferable alternative to comment banners and separators made with dashes, asterisks, or other flashy characters. If you routinely feel a need for such headings, first check that you are not trying to put too much information into a single file, then try using other alternatives to navigate the source code. Modern editors can show a summary or outline of all declarations in a file. They also allow us to add bookmarks to frequently visited parts of the code.

Exceptions Do not add a blank line between a comment header and the code it describes; this reinforces the connection between the two. Also, when there is a change in the indentation level, it is not necessary to add a blank line before the next construct because the different indent level already provides visual separation.

Alignment

R10 Do not column-align parts of statements or declarations across different lines of code. 18.2.2 p 470

Rationale Column-aligned source code might look like this:

```
architecture example of column_layout is
    signal   pc:                       std_ulogic_vector(31 downto 0) := x"0000_0000";
    signal   next_pc:                  std_ulogic_vector(31 downto 0) := x"0000_0000";
    constant INTERRUPT_VECTOR_ADDRESS: std_ulogic_vector(31 downto 0) := x"FFFF_FF80";
    signal   read:                     std_ulogic                     :=          '0';
    ...
begin
    read              <= mem_read or io_read;
    pc                <= next_pc;
    interrupt_address <= INTERRUPT_VECTOR_ADDRESS or interrupt;
    ...
end;
```

Although this is a somewhat popular style, it has many drawbacks and hinders several attributes of high-quality code:

- It is hard to write. Think about all the effort required to position each name, type and value in its own column.
- It is hard to maintain. If we rename `interrupt_address` or `INTERRUPT_VECTOR_ADDRESS`, we will break the indentation of all subsequent lines, and they will have to be manually realigned. If the names are used in other source files, we will have to search for all the occurrences and fix them.
- It is antieconomic. This kind of layout wastes a lot of space and leaves little room for actual code in each line. This increases the pressure for using names that are abbreviated or too short to tell everything about an object.
- It is not necessarily easier to read. In the declarations shown in the example, the object names and initial values are set very far apart from each other. In the signal assignments, the signal names and the corresponding value expressions are unnecessarily distant from each other.

In practice, this kind of layout is much less helpful than it seems. If you like the visual separation that it provides between object classes, names, types and values, a good alternative is to use the syntax and semantic highlighting features of modern source code editors. They allow us to specify different colors and font styles for each kind of construct in VHDL.

Line Wrapping

R11 Limit the length of a line of code to 80 characters. 18.5 p 479

Rationale Although the main reasons for this specific number are historical, it is still a good idea to use a limit that is much narrower than what current monitors can show. For instance, it allows to see two or more source files side by side, which is useful to compare different versions of a file. It allows us to read the code without horizontal scrolling. On an IDE, it allows us to use the edges of the screen for other useful information, such as a tree view of the project files on the left and a design hierarchy or source code outline on the right.

R12 If a line is longer than the specified limit, then break it at an appropriate place and continue in the next line after increasing the indentation by one level. 18.5.1 p 480

Rationale Indenting the continuation of a line makes it clear that it is the continuation of a statement or declaration, and not the beginning of a new one.

When choosing where to break the line, use the following rules[1]:

- **Break the line at a point that clearly shows it is incomplete** For instance, after a comma, a binary operator, or an opening parenthesis.

[1] This list is based on the work of Steve McConnell in *Code Complete* 2nd Ed., one of the most successful and influential books on software development. It contains hundreds of pages filled with the nitty-gritty details about the practice of writing software.

- **Keep related pieces together** This allows the reader to finish a complete thought before moving to the next line. In the example below, the lines were wrapped after each function call:

```
bounding_box := rectangle'(
    minimum(rectangle_1.top, rectangle_2.top),
    minimum(rectangle_1.left, rectangle_2.left),
    maximum(rectangle_1.bottom, rectangle_2. bottom),
    maximum(rectangle_1.right, rectangle_2. right)
);
```

- **Consider breaking down a list to one item per line** This makes it easier to see the number of items and the position of each one in the list. It also makes it easier to move the items vertically if necessary:

```
paddle <= update_sprite(
    sprite => paddle,
    sprite_x => paddle_position.x + paddle_increment.x,
    sprite_y => paddle_position.y + paddle_increment.y,
    sprite_enabled => true
);
```

- **Consider leaving the closing delimiter in a line of its own** In the previous examples, the closing parenthesis and semicolon were placed on their own line and vertically aligned with the corresponding statement, making it clear that the inner elements are enclosed between the delimiters.

If you want to make a distinction between lines that were wrapped (called *continuation lines*) and lines that were indented because of the logical structure of the code, consider adding twice the standard indentation amount before a continuation line. This complies with the rule that all indentation must be by multiples of the standard amount.

R13 Start each statement on a new line. In other words, there must be no more than one simple statement per line, and compound statements must always be broken over multiple lines. 7.4 p 143

Rationale This makes the code easier to read because we can follow the left margin and read one statement at a time; there is no need to scan the code from left to right as well. It is also easier to add, modify, reorder, delete or comment-out individual statements. When debugging, it is easier to single-step through individual comments. Finally, it makes compile errors easier to locate because each line number corresponds to only one statement.

Letter Case

R14 Choose a letter case convention for each kind of name in the code. 18.7 p 486

Rationale Besides giving the code a more consistent look, we can use the appearance of a name to convey additional information about the entity it represents. Three possible capitalization conventions are:

- *CamelCase*: consecutive words are joined without a separating character, and the beginning of each word is written in uppercase. Depending on the convention, the first letter may be in upper or lower case.
- *snake_case* or *lowercase_with_underscores*: use only lowercase letters; separate adjacent words with underscores.
- *ALL_CAPS* or *SCREAMING_SNAKE_CASE*: use only uppercase letters; separate adjacent words with underscores.

In this style guide, we use the following conventions:

- All constants and generics are named in *ALL_CAPS*;
- All other identifiers are written in *lowercase_with_underscores.*

R15 When using an acronym as part of a name, treat it as any regular word. 18.7 p 486

Rationale This prevents inconsistencies and avoids conflicts when a convention requires the first letter to be in lowercase. For instance, what should we do if our standard requires us to write variable names in lowercase but the first word is an acronym? The best way to solve this is to drop the problem altogether by treating all acronyms as

regular words. Thus, instead of `UDPHdrFromIPPacket`, we should write `UdpHdrFromIpPacket` (in CamelCase) or `udp_hdr_from_ip_packet` (in snake_case).

R16 Write all VHDL reserved words (keywords) in lowercase letters. 18.7 p 486

Rationale Text written in ALL_CAPS is too attention-grabbing, looks rough on the eyes, and is less convenient to type. It makes sense to avoid using all caps for names that appear frequently in the code, such as VHDL keywords.

2. Commenting

Comments are annotations intended to make the code easier to understand and maintain. Done right, they can save a lot of developer time and improve our productivity. Done wrong, they may duplicate information and confuse our readers. Moreover, excessive commenting is a sign that the code is too messy or hard to understand. For a detailed explanation of the basic commenting principles, a classification of the kinds of comments, and many source code examples, see chapter 17.

R17 Do not explain bad code; rewrite it! 17.2 p 430

Rationale Many comments start off on the wrong foot, when a programmer realizes that the code is becoming hard to understand. To make it "clearer," the developer adds a comment to explain the convoluted logic behind the code. Instead of using comments to explain tricky code, concentrate your effort on making it clearer and self-explanatory.

R18 Express yourself in code, not in comments. 17.2 p 430

Rationale Never settle for a comment when you can express yourself in the code proper. Among other advantages, code is always up-to-date with system behavior, whereas comments get outdated easily. Code can be debugged and verified, whereas comments cannot be tested and can hardly be trusted. Finally, comments are often duplicate information.

In the following declaration, a comment was needed only because the generic was poorly named:

```
-- Set to 1 if you want to use a hardware multiplier, 0 otherwise
generic MULTIPLIER_CONFIG: integer := 0;
```

If we choose a better name, then we can get rid of the comment:

```
 USE_HARDWARE_MULTIPLIER: boolean := false;
```

In the following condition, a comment is used to explain the author's intent:

```
-- When phase(PHASE_WIDTH-2) is 0, the current quadrant is even
if phase(PHASE_WIDTH-2) = '0' then ...
```

If we compute the condition in a variable and give it a good name, then the comment becomes unnecessary:

```
quadrant_is_even := (phase(PHASE_WIDTH-2) = '0');
if quadrant_is_even then ...
```

Always keep an eye out for opportunities to move information from comments to the code proper.

R19 Keep comments at the level of intent. 17.2 p 431

Rationale A comment written at the level of intent records the author's intention and explains the purpose of a section of code, without duplicating information. Another way to say it is that comments should tell us *why* rather than *how*; comments that tell us *how* are often redundant with the code and likely to get outdated if we change the implementation.

R20 Keep comments close to the code they describe. 17.2 p 431

Rationale This reduces the chance of a comment getting outdated; the chance that a developer will edit the code and forget to update a comment increases with the distance between them.

R21 Avoid superfluous or redundant comments. 17.3.3 p 439

Rationale Comments that only repeat what the code already says just add clutter to the code. The following comment seems to add important information because it makes it easier to understand what the statement does:

```
-- increment column counter
clctr <= clctr + 1;
```

However, the only reason why it seems useful is because the signal name was not meaningful enough. If we give the signal a good name, then the comment becomes irrelevant:

```
column_counter <= column_counter + 1;
```

R22 Use comments to document the reason behind a design decision. 17.3.1 p 432

Rationale Comments that document a design decision are an example of comments at the level of intent. In some cases, you may want to tell why you chose a particular solution from all the choices available, or why a special case is handled differently from the others. Such comments are less likely to get outdated when the code changes. They are also helpful in maintenance: another developer can compare what the code does with what it was supposed to do, making it easier to detect errors.

R23 Use comments to summarize a section or paragraph of code. 17.3.1 p 433

Rationale This helps with maintenance and increases developer productivity because it reduces the need for reading individual lines of code to grasp their overall intent.

R24 Write a documentary comment when a good name is not enough to convey all the information a reader of the code will need. 17.3.2 p 433

Rationale Sometimes the names we choose for our design units, routines, or data objects cannot convey all the necessary information. In those cases, it is useful to add a documentary comment close to their declarations. Most significant entities such as routines, packages, and other design units should be preceded by a documentary comment.

However, do not feel pressed to add such comments when they are not necessary. In many cases, a comment is unnecessary if the entity is properly named. Documentary comments should be used only when it is impossible or too cumbersome to embed all the required information in the name alone.

R25 Minimize the use of comments as separators or markers in the code. 17.3.2 p 435

Rationale Although sometimes useful, this kind of comment is usually an attempt to mitigate the real problem: the code is getting too long or complicated. Moreover, it only works if used sparingly. If the code is full of banners, each of them becomes less effective. If the markers do not add any useful information, as in the example below, they are best left out:

```
--------------------------------------------------------------------------------
-- Entity declaration
--------------------------------------------------------------------------------
entity synchronous_ram is
    --------------------------------------------------------------------------
    -- Generic declarations
    --------------------------------------------------------------------------
    generic (
        ...
```

Depending on your intention, there may be other ways to achieve the same goal without littering the code. To separate sections of code, try using one or two blank lines. If you want to be able to locate quickly certain sections of a file when skimming through the code, modern source code editors and IDEs can provide an instant outline of the source and highlight relevant places in the code hierarchy. If you want to navigate quickly to frequently visited regions of the code, many editors allow you to bookmark arbitrary points in the source files.

R26 Do not use comments to maintain version control information. 17.3.3 p 441

Rationale There are much better tools for this job. Revision control tools such as SVN, Git, and Mercurial keep a complete history of the codebase, allowing you to restore any version and undo any small change if you want. The tools can tell which files and lines were modified in each revision, with complete records of date, time, and authorship. In contrast, comments telling about changes in the source code are incomplete and inaccurate, and they grow irrelevant over time. Who knows what else was changed besides what the author decided to report in a few comments?

R27 Document the present, not the past. 17.5.3 p 449

Rationale Do not keep comments that say how things used to be done in the past. The purpose of comments is to explain why the code works now. Leave the changes history for your revision control tool.

R28 Always place the comments immediately before the code they describe. 17.2 p 431

Rationale Comments should prepare the reader for what is to follow. Because code is usually read from top to bottom, it makes sense to place the comments immediately before the code they describe. This also helps the reader to decide whether the next section of code is relevant before reading it.

R29 Avoid endline comments. Use them only sparingly. 17.4.2 p 444

Rationale Endline comments appear at the end of lines of code:

```
-- This is a full-line comment. The line consists of a comment only and no code.
/* This is also a full-line comment. */
variable minimum_value: integer; -- This is an endline comment.
variable maximum_value: integer; /* This is another endline comment.
```

This kind of comment has several problems:

- They are harder to write. When they span more than one line, you will probably want to align them vertically so that the comments do not look ragged. This requires manual alignment using tabs or spaces.
- When they span more than a line, they are wasteful of space. All lines except the first one will be mostly blank. Because there is less space left on the line, the comment will also use up more vertical space.
- They are harder to maintain. If you modify the code on the left, you will probably have to realign all the endline comments in adjacent lines.
- They leave less space for the comment text. This means that you will be hard pressed to make the comments as short as possible, instead of as clear as possible.

One of the few places where endline comments do not have as many drawbacks is to make short annotations in data declarations. In most other cases, they are best avoided.

R30 Always indent full-line comments at the same level as the code that follows. 17.4.2 p 443

Rationale This makes it clear that the comments refer to the correct level of the code. Moreover, indenting the statements and failing to do the same with comments would ruin the outline of the code, making the control structures harder to identify.

R31 Choose a low-maintenance commenting style. 17.5.1 p 445

Rationale One problem with comments is that they tend to get outdated. We can make the maintenance job easier by using comment styles that withstand modifications. This is an example that has several problems:

```
--------------------------------------------------------------------------------
-- Entity seven_segment_display_driver                                        --
-- ===================================                                        --
--                                                                            --
-- Output some text on the seven-segment displays depending on the FSM state. --
--                                                                            --
-- Inputs     Function              | Outputs     Function                    --
-- ======     ========              | =======     ========                     --
-- fsm_state  The state of the FSM  | displays    Array with 5 SSD displays   --
--------------------------------------------------------------------------------
entity seven_segment_display_driver is
    port (
        fsm_state: in state_type;
        displays: out slv_array(0 to 4)(6 downto 0)
    );
end;
```

First, it takes a lot of time to create a block comment like that. It is nicely formatted, but the same information could be communicated with a much simpler style. It is also harder to maintain. The column of dashes on the right margin needs to be realigned whenever the text changes. The comment also repeats names found in the code, making it harder to keep up-to-date. The underlining beneath the entity name will also need to be updated if the entity is renamed.

Here's a more sensible version of the same comment without any duplicate information:

```
-- Output some text on the seven-segment displays depending on the FSM state.
entity seven_segment_display_driver is
    port (
        -- The state of the FSM
        fsm_state: in state_type;
        -- Array with 5 SSD displays
        displays: out slv_array(0 to 4)(6 downto 0)
    );
end;
```

Besides being a breeze to maintain, now the comments are much closer to the code they describe.

R32 Leave a blank line before a comment header, a block comment, or a summary comment. Do not leave a blank line after these kinds of comments. 17.5.1 p 446

Rationale Comments usually indicate a new step in a train of thought, so it makes sense to separate them from the preceding code with a blank line. A blank line after the comment is unnecessary, as it would weaken the connection with the code that follows.

Exception When there is a change in indentation between the regions of code before and after the comment, a blank line is not strictly necessary because the different indent level already provides enough visual separation.

R33 Leave one whitespace between a delimiter symbol (--, /*, or */) and the comment text. 17.5.1 p 446

Rationale This makes the beginning of the comment text more clearly identifiable, especially in block comments spanning multiple lines.

R34 Write comments that are clear, correct, and concise. 17.5.2 p 446

Rationale What is the point of adding a comment if it is harder to read than the code? You do not need to write complete and grammatically correct sentences in every comment, but they need to be accurate and readable. Try not to be cryptic, and avoid unnecessary abbreviations.

R35 Do not use comments to make up for bad names. 17.5.3 p 447

Rationale Some comments only seem useful because the name of an object or routine was poorly chosen:

```
signal wr: boolean; -- true when a write has been requested
```

If we move the relevant information into the code, then the comment becomes unnecessary:

```
variable write_requested: boolean;
```

In cases like the this, the correct approach is not to write a comment, but to choose a more descriptive name.

R36 Document any surprises, workarounds, or limitations. 17.5.3 p 449

Rationale Sometimes we need to write code that is suboptimal or do things in an unusual way to circumvent a problem with a compiler or another tool. At other times, the behavior of a piece of code may be surprising at first glance. In those cases, we can save the reader some time by using a comment. It can also prevent other developers from "optimizing" or "fixing" the unusual code.

R37 Document each source code file with a header comment at the top. 17.5.4 p 450

Rationale Having a header comment in each file is good advice, provided that you follow two basic principles. First, minimize the amount of duplicate information between the code and header. Second, put in the header only information pertaining to the file as a whole. Also, do not repeat information that is better kept in the code (such as a list of all the dependencies) or in a separate file (such as the full text of a software license). Finally, do not add information that should be in a version control tool, such as the full list of authors and a history of changes. Because a file header is in a very prominent place, any noise or disinformation here is highly visible.

R38 Document each design entity, package, and routine with a header comment, unless its purpose and usage are self-evident. 17.5.4 p 453

Rationale To use an abstraction effectively, a developer must be able to treat it as a black box without peeking at its implementation. For examples of these headers, see listings 17.9 through 17.13 (pages 453-458).

R39 Avoid commenting individual statements. 17.5.4 p 458

Rationale Comments at this level of granularity should rarely be necessary. This kind of comment is usually redundant or a shallow explanation of the code. In any case, if you really need to comment an individual statement, remember to keep the information at the level of intent.

3. Declarations, Expressions, and Statements

Declarations

R40 Put each declaration on a separate line. 7.4 p 143

Rationale This makes the code easier to scan because we can follow the left margin of the code and read the declarations only from top to bottom, without needing to scan the code from left to right as well. It is also easier to add, delete, comment-out, or move a declaration without affecting the others. Finally, it makes compile errors easier to locate because each line number corresponds to only one declaration.

R41 Put each declaration in the tightest scope possible. 7.4 p 142

Rationale It is good coding practice to limit the visibility of an object to only where it is used. This makes the code safer because the object cannot be changed except in the small region where it is intended to be used. Also, if the code

does not work as expected, the region of code that needs to be debugged is smaller. Finally, it puts the declaration closer to where the object is used and avoids cluttering higher-level declarative regions in the code hierarchy.

R42 Use a consistent bit order for multibit values. 13.5 p 343

When choosing the range direction (*to* vs. *downto*) for a vector or array, follow these rules:

- If the vector represents a multibit bus, use a descending range (*downto*);
- If the vector is a collection of items or objects, use an ascending range (*to*);
- If the array is a multidimensional memory, use an ascending range for its first dimension. (i.e., the address range). For example: `type memory_type is array (0 to 1023) of std_logic_vector(31 downto 0);`

Rationale When working with bus values, we are used to seeing the most significant digit on the left, so the recommended range is (*N downto* 0). For other kinds of collections, it is more natural to use the ascending order (0 *to N*).

R43 Replace "magic numbers" with named constants. 14.1.2 p 347

Rationale "Magic numbers" are literal values that appear in the code without a clear explanation, such as:

```
status <= status xor "10010001";
if count = 96 then ...
```

Using magic numbers in the code is terrible programming practice. They make the code more enigmatic and force the readers to look for comments or figure out the meaning by themselves. They make the code harder to change because the same value may be duplicated many times in the code. This makes the changes more difficult and less reliable.

Luckily, magic numbers are easy to fix: just declare a named constant and use it in every place you use the value with the same meaning. Doing this by habit will have a tremendous impact on the quality of your source code.

Expressions

R44 Break down complicated expressions into simpler chunks. 7.4 p 144

Rationale If an expression is long, has many factors, or uses many operators, then separate it into simpler expressions and assign them to intermediate objects with meaningful names. This makes the logic of an expression easier to follow and simplifies the test when the expression is used in a condition.

R45 In expressions, use more parentheses than strictly necessary when it improves readability. 7.4 p 144 18.6 p 484

Rationale A reader should not have to remember all the precedence rules to understand an expression. By adding extra parentheses, we can make an expression more accessible to the average reader.

R46 Avoid explicit comparisons with *true* or *false*. 8.4 p 181

Rationale If the object in an equality or inequality test is of type *boolean*, then there is no need to explicitly write the values *true* or *false* in the expression. The resulting code will be cleaner and will read better in English if the values are omitted:

```
if pll_locked = true then ... -- Bad, unnecessary comparison with 'true'
if pll_locked then ...        -- Good, boolean value used directly
```

Processes

R47 Each process should have a clear goal in the design. 9.2.4 p 201

Rationale Do not group unrelated functionality into a process—it is better to have several processes that are easy to understand than a single process that no one can make sense of. Also, think about what kind of logic the process is supposed to model—combinational or sequential. The kind of logic defines several characteristics of a process, such as what should go in the sensitivity list and whether the process is allowed to keep an internal state.

R48 Each process should have a manageable size. 9.2.4 p 201

Rationale Short processes are easier to understand, debug, and maintain. As a rule of thumb, if your processes are too long to fit in one screen, then they are probably on the long side. Look for ways to make them shorter, such as moving parts of the code to separate routines.

R49 In a process, use variables whenever possible to keep local state and to calculate values that are only relevant inside the process. 9.2.4 p 202

Rationale If a task can be done completely inside the process without accessing other elements from the architecture, then do it locally.

R50 If a declaration is used only in the process, then make it local to the process. 9.2.4 p 202

Rationale The process declarative part may contain several kinds of declarations, such as constants, variables, aliases, types, subtypes, and subprograms. If one of these items is used only within a process, then there is no need to declare it in the architecture, where it would be visible to all other concurrent statements. This will keep the architecture uncluttered and each declaration closer to where it is used, improving readability.

Conditionals

R51 Avoid deeply nested structures. 10.1.1 p 224

Rationale Understanding deeply nested code is highly taxing on our brains. The code is also harder to read and write because of the long line lengths and excessive indentation. In general, deep nesting is a sign of poor coding practice. To reduce nesting, reimplement the code using simpler control logic or extract part of the code to separate routines.

R52 Replace complicated tests with one or more boolean variables. 10.2.1 p 226

Rationale A long condition expression is usually composed of two or more subconditions. A good way to make it more readable is to find pieces with a particular meaning and move them to explaining variables.

R53 Replace complicated tests with a boolean function. 10.2.1 p 227

Rationale If the condition in an *if* statement performs a meaningful test on one or more data objects, then we can extract it to a dedicated function. In this way, the test becomes simpler and much more readable.

R54 Convert nested *ifs* to a chain of *if-elsif* statements. 10.2.1 p 229

Rationale To reduce nesting, consider converting a nested sequence of *if* statements into a flat chain of *if-elsif* statements. Strictly speaking, the *if-elsif* chain is still a nested control structure. However, it reads as a series of conditions at the same depth because we can read the conditions sequentially until one of them is met.

R55 Be careful with incomplete conditionals. 10.2.1 p 230

Rationale If an object is not assigned a value during a run of the process, then it keeps its previous state. In clocked processes, this implies edge-sensitive storage or registers. In nonclocked processes, this implies latches, which are generally undesirable. For details, see "Unintentional Storage Elements (Latches or Registers)" in section 19.4.2.

R56 Try to cover all the conditions in a *case* statement without resorting to a default clause. 10.3.1 p 233

Rationale Whenever possible, try to cover all the choices explicitly without a *when others* clause. The fact that all values were covered means that the designer had to ponder about every condition. Moreover, if the *case* expression is of an enumerated type, the code will not compile if you add a value to the enumeration and forget to update the *case* statement. Had you used a *when others* clause, then the new choice would be silently absorbed—you might have introduced a bug without any warning from the compiler.

R57 Do not use the default clause to handle a normal choice value. 10.3.1 p 234

Rationale The default clause is the right place to detect errors and handle unexpected conditions, or to specify a generic course of action. However, it is not a good idea to use a *when others* clause for actions that could be associated with explicit choice values. This weakens the self-documenting nature of the *case* statement, and we lose the compiler checks for uncovered choices. In the cases where you are forced by the compiler to include a *when others* clause, consider using a *null* statement inside it.

Loops

R58 Always label nested loops if auxiliary loop control statements are used. 10.4.2 p 238

Rationale In nested loops, the *next* and *exit* statements refer to the immediately enclosing loop by default. If this is not the intended behavior, then we can label the loops and specify their identifiers after the *next* or *exit* keyword. However, because nested loops and auxiliary control statements tend to be confusing, it is a good practice to always label the loops and refer to each of them explicitly in the *next* and *exit* statements.

R59 When choosing the kind of a loop, use a *for* loop whenever possible. 10.4.4 p 242

Rationale A *for* loop controls its iteration automatically, minimizing the chance of coding mistakes. Also, it puts all the control logic in one place—at the top. This makes the loop easier to read because we do not have to look inside its body to understand the iteration logic. Use it whenever the number of iterations is known in advance.

R60 Avoid literal numbers in the loop limits. Use object attributes or a type or subtype definition whenever possible. 10.4.4 p 244

Rationale Loops that use literal values to specify ranges are hard to maintain. For instance, if we change the size of an array, then we must remember to update all loops that iterate over that array. Using a named constant solves this part of the problem, but if the number of iterations is determined by the size of an object, then it is best to use attributes such as *'range* and *'length* to define the loop limits.

R61 Choose meaningful parameter names in long or nested loops. 10.4.4 p 245

Rationale In short loops, it is customary (and even recommended) to use short parameter names, such as *i*, *j*, and *k*. However, if the loop is more than several lines long, or if it has more than a couple of levels of nesting, then try to give the loop parameters meaningful names.

4. Design Units

R62 Divide a system into as many entities and packages as needed. 6.3 p 125

Rationale A system should have as many modules as necessary, and each module should be small enough for you to reason about its entire operation at once. A good rule of thumb is to create a module for each recognizable piece of functionality that can be developed and tested separately.

Above all, do not jump through hoops to keep the entire design within a single file. Small, well-factored files are easier to understand, test, and change. They also have fewer places for bugs to hide.

R63 Give each entity or package a single, well-defined purpose. 6.3 p 124

Rationale The most important consideration when creating a module is to give it a single, well-defined responsibility. For an entity, all inputs, outputs, and behavior should support a central function. For a package, all declarations and operations should support a central purpose. If you can identify smaller, independent purposes, then move them to a separate module.

R64 Avoid grouping unrelated constants or routines in a package 14.1.2 p 349

Rationale Packages with incohesive functionality are difficult to reuse; you either end up cutting and pasting chunks of code or dragging a lot of undesired baggage. If a package starts to gather unrelated pieces of information, break it down into smaller chunks with well-defined purposes.

R65 Prefer using generics rather than constants in a package to parameterize a design entity. 6.3 p 127

Rationale Generics are the standard language feature to allow for variations in the structure of a model. With generics, we can configure two or more instances of the same entity differently. Moreover, generics are a visible part of the entity interface, making it clear that the unit is configurable. Finally, with generics, the entity can be reused in different designs without needing to copy the constant declarations.

R66 Do not use library *work*, except when meaning "the same library where the current unit is being analyzed." 6.2 p 122

Rationale The logical library name *work* is not permanently associated with any library in the host system. Rather, it acts as a pointer or an alias. During the analysis of each design unit, it gets temporarily associated with the library where the results of the current analysis will be placed. This target library is called the *working library*.

Now suppose you inadvertently create a library and give it the name *work*. How would entities in other libraries refer to it? They can't because when they use the name *work* in the code, it will point to the current working library, not to the one you have created and named *work*.

To prevent any confusion, do not give the logical name *work* to any library you create. For more on this, See the discussion on design libraries and libraries units in section 6.2, pages 121-123.

R67 Always use named association in the port map of a component instantiation.

Rationale Named association is naturally self-documenting, making the code less prone to mistakes. It is also more robust; the code that instantiates the component will not break if we change the order of the ports in the component declaration.

5. Routines

R68 Remove code duplication by moving similar sections of code into a routine. 15.2 p 373

Rationale A project that has similar code in two or more places is harder to maintain. For each modification, we must remember to update all the similar places in the code. If we forget to update one of them, we may introduce bugs that are hard to detect. Duplication also makes the code longer and harder to understand; besides having to read similar chunks of code multiple times, we must pay close attention to detect any small differences.

R69 Every routine should have a single, well-defined purpose. 15.4 p 380

Rationale Routines that do only one thing are small, easy to understand, trivial to test, and less likely to change.

R70 Write routines that take few parameters. 15.5.4 p 390

Rationale If a routine takes more than a few parameters, it is more likely to have a complicated logic and to be doing more than one thing. It is also harder to remember or figure out the role of each argument in the routine call.

R71 Consider using named association to clarify subprogram calls. 15.5.4 p 391

Rationale When a routine has more than a few parameters, or when it has an unclear interface that we are not allowed to change (e.g., when it is part of an external library), it may be a good idea to use named association instead of positional association to make the client code more readable.

R72 Put parameters in input-modify-output order. 15.5.4 p 394

Rationale To provide consistency to the readers and programmers who use your routines, establish a convention of ordering the parameters by their mode. The order input-modify-output is easy to remember because it follows the order of operations inside a routine: reading data, changing it, and outputting the results.

R73 Used unconstrained arrays as formal parameters whenever possible. 15.5.4 p 397

Rationale If a formal parameter is an unconstrained array (an array without a fixed range), then the subprogram will take the size and direction from the actual parameter. This means that you can write subprograms that are entirely generic with respect to the shape of the input arrays – for instance, using the array attributes *'range* and *'length*.

6. Names

Understanding a large piece of code is never easy, but good naming can make a big difference. If we choose names that are clear, accurate, and descriptive, then the pieces will fit together logically. If we choose names that are too cryptic, meaningless, or ambiguous, we will create systems that are nearly impossible to reason about.

Length

R74 Make a name as long as necessary. 16.1 p 402

Rationale It is a poor tradeoff to make a name shorter by removing meaningful information from it or by making it harder to read. In most cases, you should not worry about a name being too long. A long name that communicates all the important information about an entity is better than a short but cryptic one. A long name that fully describes an object is better than a short name that needs a comment to be understood.

R75 Favor clarity over brevity. 16.1 p 402

Rationale A name is clearer if it uses natural language words and describes all the important information about an object or a routine. A name is less clear if it needs to be deciphered, is unpronounceable, or uses nonstandard abbreviations. When you must choose between a clear name and one that is short, always choose clear.

R76 Use abbreviations judiciously. 16.1 p 403

Rationale Clear, unabbreviated, and pronounceable names should always be your first choice. Abbreviations make a name less pronounceable, more ambiguous, and harder to spell correctly. They are one more thing to be deciphered and remembered. Avoid abbreviating a name unless this would bring a clear and significant gain.

For the cases where an abbreviation is justifiable, here are some guidelines:

- Prefer standard abbreviations, such as the ones that can be found in a dictionary.
- Use standard or well-established abbreviations and acronyms from the hardware field, such as *clk*, *rst*, and *req*. However, make sure they cannot be mistaken for anything else. What does a name with a suffix _int mean? Is this an internal signal, an integer value, or an interrupt? The few characters saved are not worth the loss in clarity and the confusion inflicted on our readers.
- Avoid abbreviations that save only one or two letters of typing. Abbreviating *key* as *ky* and *result* as *rslt* does not justify the loss in readability.
- Use abbreviations only for names that appear frequently in the code. If a name appears occasionally, then the savings in space do not justify the loss in readability.
- Use the abbreviations consistently. Do not abbreviate the same word differently in several parts of the code. Avoid abbreviating a word in one name and then using it unabbreviated somewhere else.

General Naming Guidelines

R77 Choose meaningful names. 16.2 p 405

Rationale The name of an object or routine should contain all the important information about it. Names such as *a*, *x*, *l*, and *iv* are meaningless and force us to look somewhere else in the code. In contrast, names such as *tcp_header_length* and *interrupt_vector_address* convey all the information to communicate what the object represents. Be wary of words that only seem to have some meaning, such as *data*, *value*, or *info*; in almost every case, it is possible to find a more descriptive word.

R78 Take your time creating good names; they are too important to be chosen casually. 16.2 p 405

Rationale Good names make a huge difference in readability, which is one of the best proxies for the quality of the source code. Moreover, code is read much more often than it is written. Choosing good names takes some time, but it pays off many times over in the long run.

R79 Use clear, natural language words. 16.2 p 406

Rationale Code that reads well is easier to understand. Using natural language words helps the code read more fluently because there is no need to translate each name into something else. Read the following statements, and try to feel how the names make the code looks obvious.

```
next_count <= (current_count + 1) when count_enabled else current_count;

...

if current_position = LAST_POSITION then
    next_state <= done;
end if;
```

R80 Choose names that do not require a comment. 16.2 p 408

Rationale In most cases, the name of an object or a routine should be enough for someone else to use it with confidence. If after writing a declaration you feel it needs an explanatory comment, try choosing a more communicative name. Do not impose on the reader the burden of memorizing the actual meaning of a name.

Naming Data Objects

R81 Choose names that fully and accurately describe the entity that the object represents. 16.3 p 409

Rationale Every data object represents a piece of information in the code – the position of a switch, the state of an FSM, etc. The most important rule when naming a data object is to fully and accurately describe the entity it represents. The name should be unambiguous and immune to misinterpretation.

R82 Name things what you call them. 16.3 p 410

Rationale When choosing a name, a good way to start is to write down the same words that you would use to describe the object to someone – for example, `status_leds`, `distance_estimate_in_meters`, or `fifo_element_count`. These words will be natural, unabbreviated, and chosen to communicate meaning, which are all good attributes for a name.

R83 When naming a data object (variable, signal, constant, generic), make sure the name tells what the object holds. 16.5 p 425

Rationale Take the name INPUT_BITS, for instance. For the author, it may look perfectly clear that it contains the *number* of bits in an input port. However, this is not what the name says. According to the name, this object holds "bits." To prevent any confusion, always be explicit: if the object counts a *number* of elements, then make sure that NUMBER, NUM, or COUNT appears in the name. If it specifies a dimension, then make sure the name includes SIZE, LENGTH, or LEN.

R84 When naming data objects, use a noun or noun phrase. 16.3 p 410

Rationale Data objects represent entities manipulated in the code. Just like objects in the real world, they should be named with a noun or noun phrase. For scalar objects (objects that can hold only one instance of a value), use singular names. For array objects, use a plural name or a name that implies a collection. A plural name is usually enough to indicate that an object holds a collection of elements; most of the time, suffixes such as *_vector* or *_array* are redundant and unnecessary.

R85 Do not repeat an object's type in its name. 16.3 p 410

Rationale When you declare an object, you must provide an object class, a type, and a name. Repeating the class or type in the object's name is redundant. It makes the name longer without adding meaningful information. It also makes the name less pronounceable. And it is not really necessary: VHDL is a strongly type language, so the compiler will not let us mix up objects of different types inadvertently.

Exception When two objects represent the same entity or value and differ only in type, it may be helpful to use a suffix or prefix to differentiate between them. However, this should be done only when necessary and never by default.

R86 Avoid nondescriptive names such as *temp*, *aux*, *value*, or *flag*. 16.3 p 411

Rationale Indistinct names such as *temp*, *aux*, *value*, or *num* do not reveal anything about the entity that the object represents; it is always possible to find a more descriptive name. Other names that appear to have some meaning on the surface but need to be viewed with suspicion include *input*, *output*, *data*, and *flag*.

R87 Use prefixes and suffixes judiciously. 16.3 p 413

Rationale Like abbreviations, prefixes and suffixes can be detrimental to a name. They make a name less pronounceable; they need to be deciphered; they are one more convention that needs to be learned and documented; finally, they clutter the code, reducing the visibility of the most important part of a name—the words that convey its meaning. And, in many cases, they are just redundant information.

Some conventions prescribe a number of prefixes and suffixes on the grounds of making a name more readable, but in truth they tend to produce abominations such as `s_i_cnt_o_r_l`. If we spend more characters on redundant prefixes and suffixes than on the meaningful part of a name, then there is something wrong with our naming conventions.

How can we tell the good prefixes and suffixes from the bad ones? Good suffixes have semantic meaning and provide information that could not be inferred from the object type or class alone. Bad suffixes are redundant and just restate information that is readily visible in the object declaration. This includes the object type, the object class, or a port mode. Examples of good, semantic suffixes that do not add redundant information are:

- _n, to indicate that a signal is active low;
- _z, to indicate that a signal is part of a tri-state bus;
- _reg, to indicate that a signal is the output of a register;
- _async, to indicate that a signal has not been synchronized.

R88 Name a boolean object as an assertion or the condition of an *if* statement 16.3 p 414

Rationale An assertion is a sentence claiming that something is true, which is a good match for the nature of a boolean object. For example, `pixel_is_transparent`, `fifo_is_full`, or `output_is_ready`. Another alternative is to name boolean objects thinking about how they read as the condition of an if statement. According to this approach, `end_of_file`, `fifo_full`, or `input_available` are good naming examples.

16.3 p 415

R89 Prefer positive names for boolean objects.

Rationale Negative names make the code harder to read, especially when they need to be negated in a condition:

```
if not fifo_not_full then ...
```

Naming Routines

R90 Name a routine from the viewpoint of its user. 16.4 p 415

Rationale A developer must be allowed to call a routine trusting it to perform a task and ignoring any implementation details. Therefore, it should not be named after its implementation, but rather describe its higher-level goal. When naming a routine, always think about how it will read in the calling code.

R91 If a routine's main purpose is to return a boolean, then name it after an assertion or a question. 16.4 p 416

Rationale Functions that encapsulate a test and return *true* or *false* are called *predicate functions*. A common technique is to name this kind of function after an assertion—a statement claiming that something is true:

```
function pixel_is_transparent(pixel: pixel_type) return boolean;
function possible_to_increment(digit: bcd_digit_type) return boolean;
```

A possible variation is to name the function after a question rather than an assertion:

```
if is_pixel_transparent(x, y) then ...
if can_be_incremented(hours_digit) then ...
```

R92 If a routine's main purpose is to return an object, then name it after the return value. 16.4 p 416

Rationale The distinctive feature of a function is that it returns a value that can be used in expressions. Naming a function after its return value feels natural to the client programmer and reads well in the middle of expressions.

```
function cosine(operand: real) return real;
function bounding_box(rect_1, rect_2: rectangle_type) return rectangle_type;
function transpose(matrix: matrix_type) return matrix_type;
```

R93 If a routine's main purpose is to perform an action, then name it after everything it does. 16.4 p 418

Rationale Some routines are called for their effects on the system state: they may change objects passed as parameters or the environment in which they are executed. For such routines, choose a name that describes at a higher level of abstraction everything the routine does.

Start with a strong verb (e.g., *initialize*, *disable*, or *add*). Avoid meaningless or weak verbs such as *do*, *perform*, or *process*. Use the imperative form of the verb, and write as if you were giving a command to the routine. When the verb does not provide enough detail about what the routine does, add a noun to make the meaning perfectly clear.

Naming other VHDL constructs

R94 Name design entities using nouns or noun phrases. 16.4 p 419

Rationale Design entities are the primary hardware abstraction in VHDL. They provide the building blocks of a design, much like physical components used to build a hardware system. Therefore, it makes sense to name entities like we name things in the real world: using nouns or noun phrases. As always, avoid using names that are cryptic, are meaningless, or do not provide enough detail. Names such as `e`, `ent`, `circuit`, `design`, or `protocol` are totally meaningless.

R95 Name architectures using an adjective or adjectival phrase. 16.4 p 419

Rationale An architecture characterizes and describes the implementation of an entity; therefore, it makes sense to name it with an adjective or adjectival phrase. A common practice is to use the level of abstraction at which the architecture is implemented (e.g., `behavioral`, `rtl`, `dataflow`, `structural`, or `gate_level`). Another approach is to use a description or distinguishing feature of the implementation (e.g., `two_stage_pipeline`, `recursive`, `combinational`, or `area_optimized`).

R96 Name a design file after the design unit it contains. 16.4 p 420

Rationale To keep a design organized, each file must contain a single conceptual unit. Using the same (or a derived) name for both makes it easier to locate the file in which a given design unit is located. For an example of a file naming convention complying with this rule, see Table 16.3 on page 421.

R97 Consider naming processes with a label. 16.4 p 422

Rationale A process label documents the process intent and can provide useful debug information. However, processes exist at a lower level and in higher granularity than entities and architectures, so it may be hard to give them a good, descriptive name, especially in the form of label. Therefore, only add a label if you have something meaningful to say. Undescriptive or meaningless names are worse than no name at all.

When naming a process, use one of the following approaches:

- Name it as you would name a procedure, using the imperative form of a verb or verb phrase (e.g., `increment_pc`, `generate_outpus`, `count_events`).
- Name it as you would name an entity, using a noun or noun phrase (e.g., `next_state_logic`, `event_counter`, `input_sorter`).

Whatever the case, avoid meaningless suffixes such as _proc, _process, or _label. They do not add any meaningful information and only clutter the code.

R98 When naming a type, use a name that describes the category of objects it represents. 16.5 p 422

Rationale A type is a general category from which we create object instances. Therefore, it makes sense to use a name that describes the defining trait of the type. Following this advice, a good name for a type that represents the instructions in a CPU could be instruction, cpu_instruction, or cpu_instruction_type. Choose a name in the singular form; this is in line with the names of predefined types such as integer, real, or bit.

R99 Use a standard suffix to identify user-defined types. 16.5 p 422

Rationale In the previous example, although cpu_instruction seems like a good name for a type, it would have an undesirable consequence: the name cpu_instruction would not be available for naming signals or variables because it would be taken by the type. To prevent this problem, use a standard prefix or suffix when naming user-defined types.

A good approach is to add the suffix _type at the end of the name. Other common approaches include adding the suffix _t or the prefix t_ to the base name. In any case, do not abbreviate the word *type* by removing only the last letter, as in the suffix _typ. You get all the disadvantages of an abbreviation for the meager space savings of a single letter.

R100 When naming a type, use names that are problem-oriented rather than language- or implementation-oriented. 16.5 p 423

Rationale Instead of trying to describe a type based on its underlying representation (e.g., number of bits), choose a name that relates to the abstract kind of entity it represents. For example, for a CPU register, create a cpu_register_type rather than an unsigned_32bit_type. Also, avoid encoding the type limits or size into the name. Instead of integer_0_to_15 or word_32_bit, find out the meaning of those types in the application. Repeating the type or size information is redundant and creates an unnecessary dependency between the type's structure and its name.

R101 Name enumeration types in the singular form. 16.5 p 423

Rationale An enumerated type is not different from other scalar types such as integer or real. Therefore, like any other type, choose a name in the singular form. Note that the VHDL standard libraries use the singular form in enumerations types, such as boolean or character. Names in the singular form also read better when declaring objects of the type; the current state of an FSM is a "state", and not a "states".

R102 Name enumeration values in lowercase (snake_case). 16.5 p 424

Rationale Enumeration values are values, not named constants. Naming them in lowercase highlights this distinction.

R103 Use meaningful names for the sates of an FSM. 16.5 p 424

Rationale This saves our readers the trouble of memorizing the connection between arbitrarily chosen names and what they really represent. Avoid meaningless names such as S1, S2, and S3. For example, in a garage door, we could use opened, closing, closed, and opening. In a CPU, we could use fetch, decode, and execute, and so on.

R104 When applicable, use suffixes with semantic meaning for naming signals. However, avoid suffixes that only repeat information provided with the signal declaration. 16.5 p 426

Rationale In some cases, it may be useful to use a signal's name to provide additional information about its role in a design. This can be done by attaching a semantic suffix to the signal name. Some examples include:

- _n to indicate that the signal is active low;
- _z to indicate that the signal is part of a tri-state bus;
- _reg to indicate that the signal is the output of a register;
- _async to indicate that the signal has not been synchronized.

These kinds of suffixes are important because they register our intent and add information that is not present elsewhere in the code. In contrast, prefixes and suffixes that merely repeat information provided at the signal declaration (such as its type) are redundant and should be avoided.

R105 Document all the suffixes and abbreviations used in the design in a project-wide file. 16.1 p 403

Rationale This documents our intent and prevents developers from using different suffixes or abbreviations with the same meaning. Place these files under version control together with the source code, and always keep them up to date.

R106 When naming ports, use meaningful and communicative names that provide most of the information needed to instantiate and use it correctly. 16.5 p 427

Rationale The more effort we put into choosing descriptive port names, the easier the entity will be to use. It also minimizes the amount of guesswork required from the reader. If we use meaningless names such as a, b, c, x, and y, we leave our readers without a clue and they must figure out the code on their own.

R107 For simple loops spanning few lines of code, it is all right to use the customary single-letter names *i*, *j*, and *k* for the loop parameter. 16.5 p 428

Rationale Although these names are not communicative, they are in line with the practice of choosing the length of a name in proportion to its scope. If the loops are short, then the loop header is always within view, and it is unlikely that the loop index could be mistaken for something else. Also, loop parameters are commonly used as array indices and in other expressions, often several times in a single line of code. This increases the need for shorter names. Finally, these identifiers have been used for loop indices since Fortran, so they are an established convention. Most developers will readily identify the names *i*, *j*, and *k* as loop indices. However, they should be avoided if the loop is longer than a few lines of code, if it is nested, or if its logic is complex. In such cases, consider using names that refer to what the loop is iterating over, such as *horizontal_offset* or *row_number*.